Hugues Bersini

6e édition

La programmation orientée objet

Cours et exercices en UML 2
avec Java, C#, C++, Python, PHP et LINQ

EYROLLES

 ISBN : 978-2-212-13578-7

Avant-propos

Aux tout débuts de l'informatique, le fonctionnement interne des processeurs décidait de la seule manière efficace de programmer un ordinateur. Alors que l'on acceptait tout programme comme une suite logique d'instructions, il était admis que l'organisation du programme et la nature même de ces instructions ne pouvaient s'éloigner de la façon dont le processeur les exécutait : pour l'essentiel, des modifications de données mémorisées, des déplacements de ces données d'un emplacement mémoire à un autre, des opérations d'arithmétique et de logique élémentaire, et de possibles ruptures de séquence ou branchements.

La mise au point d'algorithmes complexes, dépassant les simples opérations mathématiques et les simples opérations de stockage et de récupérations de données, obligea les informaticiens à effectuer un premier saut dans l'abstrait, en inventant un style de langage dit procédural, auquel appartiennent les langages Fortran, Cobol, Basic, Pascal, C... Les codes écrits dans ces langages sont devenus indépendants des instructions élémentaires propres à chaque type de processeur. Grâce à eux, les informaticiens ont pris quelques distances par rapport aux processeurs (en ne travaillant plus directement à partir des adresses mémoire et en évitant la manipulation directe des instructions élémentaires) et ont élaboré une écriture de programmes plus proche de la manière naturelle de poser et de résoudre les problèmes. Il est incontestablement plus simple d'écrire : c = a + b qu'une suite d'instructions telles que `"load a, reg1"`, `"load b, reg2"`, `"add reg3, reg1, reg2"`, `"move c, reg3"`, ayant pourtant la même finalité. Une opération de traduction automatique, dite de compilation, se charge de traduire le programme, écrit au départ avec ces nouveaux langages, dans les instructions élémentaires seules comprises par le processeur.

La montée en abstraction permise par ces langages de programmation présente un double avantage : une facilité d'écriture et de résolution algorithmique, ainsi qu'une indépendance accrue par rapport aux différents types de processeur existant aujourd'hui sur le marché. Le programmeur se trouve libéré des détails d'implémentation machine et peut se concentrer sur la logique du problème et ses voies de résolution.

Plus les problèmes à affronter gagnaient en complexité – comptabilité, jeux automatiques, compréhension et traduction des langues naturelles, aide à la décision, bureautique, conception et enseignement assistés, programmes graphiques, etc. –, plus l'architecture et le fonctionnement des processeurs semblaient contraignants. Il devenait vital d'inventer des mécanismes informatiques simples à mettre en œuvre pour réduire cette complexité et rapprocher encore plus l'écriture de programmes des manières humaines de poser et résoudre les problèmes.

Avec l'intelligence artificielle, l'informatique s'inspira de notre mode cognitif d'organisation des connaissances, comme un ensemble d'objets conceptuels entrant dans un réseau de dépendance et se structurant de manière taxonomique. Avec la systémique ou la bioinformatique, l'informatique nous révéla qu'un ensemble d'agents au fonctionnement élémentaire, mais s'influençant mutuellement, peut produire un comportement émergent d'une surprenante complexité lorsqu'on observe le système dans sa globalité. Dès lors, pour comprendre jusqu'à reproduire ce comportement par le biais informatique, la meilleure approche consiste en une découpe adéquate du système en ses parties et une attention limitée au fonctionnement de chacune d'entre elles.

Tout cela mis ensemble (la nécessaire distanciation par rapport au fonctionnement du processeur, la volonté de rapprocher la programmation du mode cognitif de résolution de problème, les percées de l'intelligence artificielle et de la bio-informatique, le découpage comme voie de simplification des systèmes apparemment complexes) conduisit graduellement à un deuxième type de langages de programmation, fêtant ses 45 ans d'existence (l'antiquité à l'échelle informatique) : les langages orientés objet, tels Simula, Smalltalk, C++, Eiffel, Java, C#, Delphi, Power Builder, Python et bien d'autres...

L'orientation objet (OO) en quelques mots

À la différence de la programmation procédurale, un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Chaque objet se décrit par un ensemble d'attributs (partie statique) et de méthodes portant sur ces attributs (partie dynamique). Certains de ces attributs étant l'adresse des objets avec lesquels les premiers coopèrent, il leur est possible de déléguer certaines des tâches à leurs collaborateurs. Le tout s'opère en respectant un principe de distribution des responsabilités on ne peut plus simple, chaque objet s'occupant de ses propres attributs. Lorsqu'un objet exige de s'informer sur les attributs d'un autre ou de les modifier, il charge cet autre de s'acquitter de cette tâche. En effet, chaque objet expose à ses interlocuteurs un mode d'emploi restreint, une carte de visite limitée aux seuls services qu'il est apte à assurer et continuera à rendre dans le temps, malgré de possibles modifications dans la réalisation concrète de ces services.

Cette programmation est fondamentalement distribuée, modularisée et décentralisée. Pour autant qu'elle respecte également des principes de confinement et d'accès limité (dits d'encapsulation, l'objet n'expose qu'une partie restreinte de ses services), cette répartition modulaire a également l'insigne avantage de favoriser la stabilité des développements. En effet, elle restreint au maximum les conséquences de modifications apportées au code au cours du temps : seuls les objets concernés sont modifiés, pas leurs interlocuteurs, même si le comportement de ces derniers dépend indirectement des fonctionnalités affectées.

Ces améliorations, résultant de la prise de conscience des problèmes posés par l'industrie du logiciel (complexité accrue et stabilité dégradée), ont enrichi la syntaxe des langages objet. Un autre mécanisme de modularisation inhérent à l'orienté objet est l'héritage, qui permet à la programmation de refléter l'organisation taxonomique de notre connaissance en une hiérarchie de concepts du plus au moins général. À nouveau, cette organisation modulaire en objets génériques et plus spécialistes est à

l'origine d'une simplification de la programmation, d'une économie d'écriture et de la création de zones de code aux modifications confinées. Tant cet héritage que la répartition des tâches entre les objets autorisent une décomposition plus naturelle des problèmes, une réutilisation facilitée des codes déjà existants (tout module peut se prêter à plusieurs assemblages) et une maintenance facilitée et allégée de ces derniers. L'orientation objet s'impose, non pas comme une panacée universelle, mais comme une évolution naturelle de la programmation procédurale qui facilite l'écriture de programmes, les rendant plus gérables, plus compréhensibles, plus stables et mieux réexploitables.

L'orienté objet inscrit la programmation dans une démarche somme toute très classique pour affronter la complexité de quelque problème qui soit : une découpe naturelle et intuitive en des parties plus simples. A fortiori, cette découpe sera d'autant plus intuitive qu'elle s'inspire de notre manière « cognitive » de découper la réalité qui nous entoure. L'héritage, reflet fidèle de notre organisation cognitive, en est le témoignage le plus éclatant. L'approche procédurale rendait cette découpe moins naturelle, plus « forcée ». Si de nombreux adeptes de la programmation procédurale sont en effet conscients qu'une manière incontournable de simplifier le développement d'un programme complexe est de le découper physiquement, ils souffrent de l'absence d'une prise en compte naturelle et syntaxique de cette découpe dans les langages de programmation utilisés. Dans un programme imposant, l'OO aide à tracer les pointillés que les ciseaux doivent suivre là où il semble le plus naturel de les tracer : au niveau du cou, des épaules ou de la ceinture, et non pas au niveau des sourcils, des biceps ou des mollets. De surcroît, cette pratique de la programmation incite à cette découpe suivant deux dimensions orthogonales : horizontalement, les classes se déléguant mutuellement un ensemble de services, verticalement, les classes héritant entre elles d'attributs et de méthodes installés à différents niveaux d'une hiérarchie taxonomique. Pour chacune de ces dimensions, reproduisant fidèlement nos mécanismes cognitifs de conceptualisation, en plus de simplifier l'écriture des codes, il est important de faciliter la récupération de ces parties dans de nouveaux contextes et d'assurer la robustesse de ces parties aux changements survenus dans d'autres. Un code OO, idéalement, sera aussi simple à créer qu'à maintenir, récupérer et faire évoluer.

Il n'est pas pertinent d'opposer le procédural à l'OO car, in fine, toute programmation des méthodes (c'est-à-dire la partie active des classes et des objets) reste totalement tributaire des mécanismes procéduraux. On y rencontre des variables, des arguments, des boucles, des fonctions et leurs paramètres, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils procédurales. L'OO vient plutôt compléter le procédural, en lui superposant un système de découpe plus naturel et facile à mettre en œuvre. Pour preuve, les langages procéduraux comme le C, Cobol ou, plus récemment, PHP, se sont relativement aisément enrichis d'une couche dite OO sans que cette addition ne remette sérieusement en question l'existant.

Cependant, l'effet de cette couche additionnelle ne se limite pas à quelques structures de données supplémentaires afin de mieux organiser les informations manipulées par le programme. Il va bien au-delà. C'est toute une manière de concevoir un programme et la répartition de ses parties fonctionnelles qui est en jeu. Les fonctions et les données ne sont plus d'un seul tenant mais éclatées en un ensemble de modules reprenant chacun à son compte une sous-partie de ces données et les seules fonctions qui les manipulent. Il faut réapprendre à programmer en s'essayant au développement d'une succession de micro-programmes et au couplage soigné et réduit au minimum de ces micro-programmes.

En découpant 1 000 lignes de code en 10 modules de 100 lignes, le gain est bien plus que linéaire, car il est extraordinairement plus simple de programmer 100 lignes plutôt que 1 000. En subs-

tance, la programmation OO pourrait reprendre à son compte ce slogan altermondialiste : « agir localement, penser globalement ».

Se pose alors la question de tactique didactique, très controversée dans l'enseignement de l'informatique aujourd'hui, sur l'ordre dans lequel enseigner procédural et OO. De nombreux enseignants, soutenus en cela par de très nombreux manuels, considèrent qu'il faut d'abord passer par un enseignement intensif et une maîtrise parfaite du procédural, avant de faire le grand saut vers l'OO. Mais vingt années d'enseignement de la programmation à des étudiants de tous âges et de toutes conditions (issus des sciences humaines ou exactes) nous ont convaincus qu'il n'y a aucun ordre à donner. De même qu'historiquement, l'OO est né quasiment en même temps que le procédural et en complément de celui-ci, l'OO doit s'enseigner conjointement et en complément du procédural. Il faut enseigner les instructions de contrôle en même temps que la découpe en classes. L'enseignement de la programmation doit mélanger à loisir la perception « micro » des mécanismes procéduraux à la vision « macro » de la découpe en objets. Aujourd'hui, tout projet informatique de dimension conséquente débute par une analyse des différentes classes qui le constituent. Il faut aborder l'enseignement de la programmation tout comme débute la prise en charge de ce type de projet, en enseignant au plus vite la manière dont ces classes et les objets qui en résultent opèrent à l'intérieur d'un programme.

Ces dernières années, compétition oblige, l'orienté objet s'est trouvé à l'origine d'une explosion de technologies différentes, mais toutes intégrant à leur manière ses mécanismes de base : classes, objets, envois de messages, héritage, encapsulation, polymorphisme... Ainsi sont apparus de nombreux langages de programmation proposant des syntaxes dont les différences sont soit purement cosmétiques, soit plus subtiles. Ils sont autant de variations sur les thèmes créés par leurs trois principaux précurseurs : Simula, Smalltalk et C++.

L'OO a également conduit à repenser trois des chapitres les plus importants de l'informatique de ces deux dernières décennies :

- tout d'abord, le besoin de développer une méthode de modélisation graphique standardisée débouchant sur un niveau d'abstraction encore supplémentaire (on ne programme plus en écrivant du code mais en dessinant un ensemble de diagrammes, le code étant créé automatiquement à partir de ceux-ci ; c'est le rôle joué par UML 2) ;
- ensuite, les applications informatiques distribuées (on ne parlera plus d'applications distribuées mais d'objets distribués, et non plus d'appels distants de procédures mais d'envois de messages à travers le réseau) ;
- enfin, le stockage des données, qui doit maintenant compter avec les objets.

Chaque fois, plus qu'un changement de vocabulaire, un changement de mentalité sinon de culture s'impose.

Les grands acteurs de l'orienté objet

Aujourd'hui, l'OO est omniprésent. Microsoft par exemple, a développé un nouveau langage informatique purement objet (C#), a très intensément contribué au développement d'un système d'informatique distribuée, basé sur des envois de messages d'ordinateur à ordinateur (les services web) et a plus

récemment proposé un nouveau langage d'interrogation des objets (LINQ), qui s'interface naturellement avec le monde relationnel et le monde XML. Tous les langages informatiques intégrés dans sa nouvelle plate-forme de développement, .Net (aux dernières nouvelles, ils seraient 22), visent à une uniformisation (y compris les nouvelles versions de Visual Basic et Visual C++) en intégrant les mêmes briques de base de l'OO. Aboutissement considérable s'il en est, il devient très simple de faire communiquer ou hériter entre elles des classes écrites dans des langages différents.

Plusieurs années auparavant, Sun avait conçu Java, une création déterminante car elle fut à l'origine de ce nouvel engouement pour une manière de programmer qui pourtant existait depuis toujours sans que les informaticiens dans leur ensemble en reconnaissent l'utilité ni la pertinence. Depuis, Sun a créé RMI, Jini et sa propre version des services web, tous basés sur les technologies OO. Ces mêmes services web font l'objet de développements tout autant aboutis chez HP ou IBM. À la croisée de Java et du Web (originellement la raison, sinon du développement de Java, du moins de son succès), on découvre une importante panoplie d'outils de développement et de conception de sites web dynamiques. Depuis, Java est devenu le langage de prédilection pour de nombreuses applications d'entreprise et plus récemment pour le développement d'applications tournant sur les smartphones et tablettes dotés du système Android, maintenu par Google.

IBM et Borland, avec Rational et Together, menaient la danse en matière d'outils d'analyse du développement logiciel, avec la mise au point de puissants environnements UML. Chez IBM, la plate-forme logicielle Eclipse est sans doute, à ce jour, une des aventures Open Source les plus abouties en matière d'OO. Comme environnement de développement Java, Eclipse est aujourd'hui le plus prisé et le plus usité et cherche à gagner son pari « d'éclipser » tous les autres. Borland a rendu Together intégrable tant dans Visual Studio.Net que dans Eclipse, comme outil synchronisant au mieux et au plus la programmation et la réalisation des diagrammes UML.

Enfin, l'OMG, organisme de standardisation du monde logiciel, n'a pas pour rien la lettre O comme initiale. UML et Corba sont ses premières productions : la version OO de l'analyse logicielle et la version OO de l'informatique distribuée. Cet organisme plaide de plus en plus pour un développement informatique détaché des langages de programmation ainsi que des plates-formes matérielles, par l'utilisation intensive des diagrammes UML. Partant de ces mêmes diagrammes, les codes seraient créés automatiquement dans un langage choisi et en adéquation avec la technologie voulue.

Le pari d'UML est osé et encore très largement controversé, mais l'évolution de l'informatique au cours des ans a toujours confié à des mécanismes automatisés le soin de prendre en charge des détails qui éloignaient le programmeur de sa mission première : penser et résoudre son problème.

Objectifs de l'ouvrage

Toute pratique économe, fiable et élégante de Java, C++, C#, Python, PHP ou UML requiert, pour débuter, une bonne maîtrise des mécanismes de base de l'OO. Et, pour y pourvoir, rien n'est mieux que d'expérimenter les technologies OO dans ces différentes versions, comme un bon conducteur qui se sera frotté à plusieurs types de véhicules, un bon skieur à plusieurs styles de skis et un guitariste à plusieurs modèles de guitares.

Plutôt qu'un voyage en profondeur dans l'un ou l'autre de ces multiples territoires, ce livre vous propose d'explorer plusieurs d'entre eux, mais en tentant à chaque fois de dévoiler ce qu'ils recèlent de commun. Car ce sont ces ressemblances qui constituent les briques fondamentales de l'OO. Nous pensons que la mise en parallèle de C++, Java, C#, Python, PHP 5 et UML est une voie privilégiée pour l'extraction de ces mécanismes de base.

Il nous a paru pour cette raison indispensable de discuter et comparer la façon dont ces cinq langages de programmation gèrent, par exemple, l'occupation mémoire par les objets, leur manière d'implémenter le polymorphisme ou la programmation dite « générique », pour en comprendre in fine toute la problématique et les subtilités indépendamment de l'une ou l'autre implémentation. Ajoutez une couche d'abstraction, ainsi que le permet UML, et cette compréhension ne pourra que s'en trouver renforcée. Chacun de ces cinq langages offre des particularités amenant les praticiens de l'un ou l'autre à le prétendre supérieur aux autres : la puissance du C++, la compatibilité Windows et l'intégration XML de C#, l'anti-Microsoft et le leadership de Java en matière de serveurs d'entreprise, les vertus pédagogiques et l'aspect « scripting » de Python, le succès incontestable de PHP 5 pour la mise en place simplifiée d'une solution web dynamique et la capacité de s'interfacer aisément avec les bases de données. Nous nous désintéresserons ici complètement de ces querelles de clochers, a fortiori car notre projet pédagogique nous conduit bien davantage à nous pencher sur ce qui les réunit plutôt que ce qui les différencie. C'est leur multiplicité qui a présidé à cet ouvrage et qui en fait, nous l'espérons, son originalité. Nous n'allons pas nous en plaindre et défendons en revanche l'idée que le choix définitif de l'un ou l'autre de ces langages dépend davantage d'habitude, d'environnement professionnel ou d'enseignement, de questions sociales et économiques et surtout de la raison concrète de cette utilisation (pédagogie, performance machine, adéquation web ou base de données …).

Quelques amabilités glanées dans *Masterminds of Programming*

Bjarne Stroustrup (créateur du C++) : « *J'avais prédit que s'il voulait percer, Java serait contraint de croître significativement en taille et en complexité. Il l'a fait* ».

Guido van Rossum (créateur de Python) : « *Je dis qu'une ligne de Python, de Ruby, de Perl ou de PHP équivaut à 10 lignes de Java* ».

Tom Love (co-créateur d'Objective-C, le langage OO de prédilection pour le développement des applications Apple et plus récemment IPhone, IPod et autres smartphones) : « *Tant Objective-C que C++ sont nés au départ du langage C. Dans le premier cas, ce fut un petit langage, simple, élégant, net et bien défini ; dans l'autre, ce fut une abomination hyper compliquée et présentant de véritables défauts de conceptions* ».

James Gosling (créateur de Java) : « *Les pointeurs en C++ sont un désastre, une véritable incitation à programmer de manière erronée* » et « *C# a tout pompé sur Java, à l'exception de la sûreté et de la fiabilité par la prise en charge de pointeurs dangereux qui m'apparaissent comme grotesquement stupides* ».

Anders Hejlsberg (créateur de C#) : « *Je ne comprends pas pourquoi Java a choisi de ne pas évoluer. Si vous regardez l'histoire de l'industrie, tout n'est qu'une question d'évolution. À la minute où vous arrêtez d'évoluer, vous signez votre arrêt de mort* ».

James Rumbaugh (un des trois concepteurs d'UML) : « *Je pense qu'utiliser UML comme générateur de code est une idée exécrable. Il n'y a rien de magique au sujet d'UML. Si vous pouvez créer du code à partir des diagrammes, alors il s'agit d'un langage de programmation. Or UML n'est pas du tout conçu comme un langage de programmation* ».

Bertrand Meyer (créateur d'Eiffel et défendant la programmation OO dite par « contrats ») : « *Je ne comprends pas comment l'on peut programmer sans prendre le temps et la responsabilité de se demander ce que les éléments du programme ont la charge de faire. C'est une question à poser à Gosling, Stroustrup, Alan Kay ou Hejlsberg* ».

Récemment, un livre intitulé *Masterminds of Programming* (O'Reilly, 2009) et compilant un ensemble d'entretiens avec les créateurs des langages de programmation, nous a convaincu du bien-fondé de ce type de démarche comparative. Il apparaît en effet, au vu de la guerre des mots à laquelle se livrent ses créateurs, qu'aucun langage de programmation ne peut vraiment s'appréhender sans partiellement le comparer à d'autres. En fait, tous se positionnent dans une forme de rupture ou de replacement par rapport à ses prédécesseurs.

Enfin, nous souhaitions que cet ouvrage, tout en restant suffisamment détaché de toute technologie, couvre l'essentiel des problèmes posés par la mise en œuvre des objets en informatique : leur stockage sur le disque dur, leur interfaçage avec les bases de données, leur fonctionnement en parallèle et leur communication à travers Internet. Ce faisant nous acceptons de perdre un peu en précision, et il nous apparaît nécessaire de mettre en garde le lecteur. Ce livre n'aborde aucune des technologies en profondeur, mais les approche toutes dans ce qu'elles ont de commun et qui se devrait de survivre pour des siècles et des siècles…

Plan de l'ouvrage

Les 23 chapitres de ce livre peuvent se répartir en cinq grandes parties.

Le **premier chapitre** constitue une partie en soi car il a pour importante mission d'initier aux briques de base de la programmation orientée objet, sans aucun développement technique : une première esquisse, teintée de sciences cognitives, et toute en intuition, des éléments essentiels de la pratique OO.

La **deuxième partie** intègre les quatorze chapitres suivants. Il s'agit pour chacun d'entre eux de décrire, plus techniquement cette fois, ces briques de base : objet, classe (**chapitres 2 et 3**), messages et communication entre objets (**chapitres 4, 5 et 6**), encapsulation (**chapitres 7 et 8**), gestion mémoire des objets (**chapitre 9**), modélisation objet (**chapitre 10**), héritage et polymorphisme (**chapitres 11 et 12**), classe abstraite (**chapitre 13**), clonage et comparaison d'objets (**chapitre 14**), interface (**chapitre 15**).

Chacune de ces briques est illustrée par des exemples en Java, C#, C++, Python, PHP 5 et UML. Nous y faisons le pari que cette mise en parallèle est la voie la plus naturelle pour la compréhension des mécanismes de base : extraction du concept par la multiplication des exemples.

La **troisième partie** reprend, dans le droit fil des ouvrages dédiés à l'un ou l'autre langage objet, des notions jugées plus avancées : les objets distribués, Corba, RMI, les services web (**chapitre 16**), le multithreading ou programmation parallèle (ou concurrentielle, **chapitre 17**), la programmation événementielle (**chapitre 18**) et enfin la sauvegarde des objets sur le disque dur, y compris l'interfaçage entre les objets et les bases de données relationnelles (**chapitre 19**). Là encore, le lecteur se trouvera le plus souvent en présence de plusieurs versions dans les cinq langages de ces mécanismes. La nouvelle plate-forme LINQ de Microsoft s'y trouve abordée, car elle propose une manière radicalement neuve de penser la mise en correspondance entre les objets et les informations stockées sous forme relationnelle ou XML.

La **quatrième partie** décrit plusieurs projets de programmation objet totalement aboutis, tant en UML qu'en Java. Elle inclut d'abord le **chapitre 20**, décrivant la modélisation objet d'un petit flipper et d'un

petit tennis, ainsi que les problèmes de conception orientée objet que cette modélisation pose. Le **chapitre 21**, lié au **chapitre 22**, décrit la manière dont les objets peuvent s'organiser en liste liée ou en graphe, mode de mise en relation et de regroupement des objets que l'on retrouve abondamment dans toute l'informatique. Le **chapitre 22** marie la chimie, la biologie et l'économie à la programmation OO. Il contient tout d'abord la programmation d'un réacteur chimique créant de nouvelles molécules à partir de molécules de base, tout en suivant à la trace l'évolution de la concentration des molécules dans le temps. La chimie – une chimie élémentaire acquise bien avant l'université – nous est apparue être une plate-forme pédagogique idéale pour l'assimilation des concepts objets. Nous proposons aussi des simulations élémentaires du système immunitaire et d'une économie de marché.

Enfin, la **dernière partie** se ramène au seul **chapitre 23**, dans lequel sont présentées plusieurs recettes de conception OO, résolvant de manière fort élégante un ensemble de problèmes récurrents dans la réalisation de programmes. Ces recettes de conception, dénommées *Design patterns*, sont devenues fort célèbres dans la communauté OO. Leur compréhension s'inscrit dans la suite logique de l'enseignement des briques et des mécanismes de base de l'OO. Elle fait souvent la différence entre l'apprenti et le compagnon parmi les programmeurs OO. Nous les illustrons en partie sur le flipper, la chimie, la biologie et l'économie des chapitres précédents.

À qui s'adresse ce livre ?

Cet ouvrage est sans nul doute destiné à un public assez large : industriels, enseignants et étudiants, mais sa vocation première n'en reste pas moins une initiation à la programmation orientée objet.

Ce livre sera un compagnon d'étude enrichissant pour les étudiants qui comptent la programmation objet dans leur cursus d'étude (et toutes technologies s'y rapportant : Java, C++, C#, Python, PHP, Corba, RMI, services web, UML, LINQ). Il devrait les aider, le cas échéant, à évoluer de la programmation procédurale à la programmation objet, pour aller ensuite vers toutes les technologies s'y rapportant.

Nous ne pensons pas, en revanche, que ce livre peut seul prétendre à constituer une porte d'entrée dans le monde de la programmation tout court. Comme dit précédemment, nous estimons qu'il est idéal d'aborder en même temps les mécanismes OO et procéduraux. Pour des raisons évidentes de place et car les librairies informatiques en regorgent déjà, nous avons fait l'impasse d'un enseignement de base des mécanismes procéduraux : variables, boucles, instructions conditionnelles, éléments fondamentaux et compagnons indispensables à l'assimilation de l'OO. Nous pensons, dès lors, que ce livre sera plus facile à aborder pour des lecteurs ayant déjà un peu de pratique de la programmation dite procédurale, et ce, dans un quelconque langage de programmation. Finalement, précisons que s'il ne pretend pas être exhaustif – et bien qu'à sa 6e édition – il résiste assez bien au temps. Ses pages ne jaunissent pas trop et, tout comme la plupart des technologies qu'il recense, il reste rétrocompatible avec ses versions précédentes. Il suit les évolutions de toutes ces technologies, même si celles-ci restent délibérément accrochées aux principes OO qui en font son sujet et, nous l'espérons, son attrait principal.

Table des matières

Index